Project A

CNN-Based Cell Counting

In this project you get to try using convolutional neural networks for counting blood cells in microscopy images. You can start from the code that you used for Lab 2, but you will have to modify it to fit your needs.

On ping-pong, you will find 50 images with blood cells and 50 mat-files with manually selected blood cell centres. The finished algorithm will be tested on a set of similar images that you don't get to see beforehand.

A.1 Training

To train a detector you need to extract positive and negative examples from the training images. The more different examples you can generate the better. Hence a good strategy is to extract a few thousand examples, train one epoch using these examples and then extract new examples for the next epoch.

- Write code to extract positive and negative examples and train a network for cell detection.
- Design an appropriate network for this problem. Use a validation set to compare different network structures.

A.2 Applying your detector

To evaluate how well you are doing, it's a good idea to perform sliding window detection on validation images. To do that you can use the provided `sliding_cnn.m`. You need to tell it what stride to use. You can use stride 4 to speed up the process. Use thresholding together with non-maximum suppression to get the final cell detections.

```
probmap = sliding_cnn(net, img, stride);
```

By default this function will pad the image with zeros at the borders so you get detections even very close to borders, but feel free to modify this function so it suits your needs. Note that due to the stride, the output `probmap` has fewer pixels than the input `img`. Exactly how to go from coordinates in `probmap` to coordinates in `img` depends on how you did the training, but the following rule should be roughly correct,

```
x  = (xp - 1) * stride + 1
```

where `x` refers to a coordinate in `img` and `xp` to a coordinate in `probmap`. The same formula holds for both the y coordinates.

A.3 Bonus level

For the bonus level, you need to perform the following extra tasks

1. Train a fully-convolutional network.
2. Use data augmentation (rotation, scaling, small tranlslation and intensity variation) when you extract examples.
3. A loop to generate and train on hard examples.
4. Sub-pixel precision when computing the detections.
5. Be prepared to explain your solution individually.

Fully-convolutional network

Train a fully-convolutional network using small input *patches* (where the output is a single pixel) and then use it on full-size images. To avoid very slow execution, it is recommended that you use two max-pooling layers so the output has 1/16 the number of pixels as the input.

Matlab R2016b. Unfortunately, Matlab R2016b has no support for fully-convolutional networks. You can apply your network using `sliding_cnn`. Choose `stride` according to the total stride in your network. For example, if you have two max pooling layers with stride 2 and a convolutional layer with stride 2, you have a total stride of 8.

Matlab R2017b. When defining the network on R2017b, use `pixelClassificationLayer` rather than `classificationLayer`. Also avoid normalization of the input by setting

```
imageInputLayer(..., 'Normalization', 'none')
```

when you define the network. When applying your network to a full-sized image, you can use the provided

```
probs = sliding_fcn(net, img);
```

It applies the fully-convolutional network on the whole input image at once. Note that this will not work unless your network is fully-convolutional. Feel free to modify this function if it doesn't fit with your network.

Data augmentation

Just 50 images is not a huge dataset so it is a good idea to extend the dataset using data augmentation. You can look at `affine_warp` from Lab 3 for inspiration.

Finding hard examples

A problem with completely random negative examples is that they rarely hit really difficult examples. To improve the detector further we can actively search for these hard examples and focus on improving on these. To do that, your current detector on the images in the training set and store all the errors that you make in a list. Add these points to your dataset structure. Now each time you pick a sample, select one of these *hard examples* with a certain probability (not too large). This will focus the training on these hard examples.

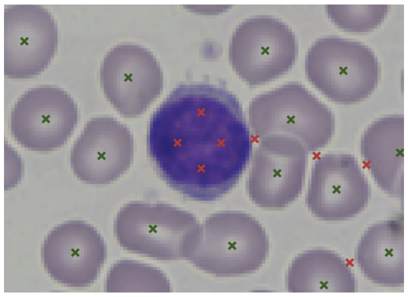

Figure A.1: An example of correct (green) and incorrect (red) detections from a detector that has been trained without hard examples. If we focus the training more on the red points, the detector will hopefully improve.

Sub-pixel precision

To get a fast detector we use max pooling (or convolutional) layers with stride= 2. Naturally this makes the precision in the output detections lower. Hence we want to compute the detections with sub-pixel precision (as described earlier in the course). You might want to use Gaussian smoothing before applying the sub-pixel algorithm.

A.4 What to hand in

Hand in your code along with a partial report; see below. Your detectors will be evaluated on a set of new images. To simplify this, provide a matlab file

```
detections = run_detector(img)
```

that runs your detector on the image `img`. You can assume that `img` is a color image with values between 0 and 1. For `run_detector.m` to be able to run your network it is recommended to save the network using

```
save('my_network.mat', 'net')
```

and then load it in `run_detector.m`

A small prize for the best detector will be awarded.

Poster. Make a scientific poster describing your project. Try to keep the amount of text down and use a lot of images instead. Give a very short overview of the full project but focus on the most interesting part. Make sure that you somehow validate how well your network works.